Research Article

FAULT TREE ANALYSIS FOR UML (UNIFIED MODELING LANGUAGE)

[1]Supriya Shivhare, Prof. Naveen Hemranjani

Address for Correspondence

[1]Student, M.Tech (S.E.)
[2]Vice Principal (M.Tech)
Suresh Gyan Vihar University, Jaipur(Raj.), India

ABSTRACT

Software plays an increasing role in the safety critical systems. Increasing the quality and reliability of the software has become the major objective of software development industry. Researchers and industry practitioners, look for innovative techniques and methodologies that could be used to increase their confidence in the software reliability. Fault tree analysis (FTA) is one method under study at the Software Assurance Technology Center (SATC) of NASA's Goddard Space Flight Center to determine its relevance to increasing the quality and the reliability of software. This paper briefly reviews some of the previous research in the area of Software Fault Tree Analysis (SFTA). Next we discuss a roadmap for application of the SFTA to software, with special emphasis on object-oriented design. This is followed by a brief discussion of the paradigm for transforming a software design artifact (i.e., sequence diagram) to its corresponding software fault tree.

1. INTRODUCTION

Fault Tree Analysis (FTA) [1] is a technique used in the area of reliability. Initially, FTA was introduced in the 1960s, with the primary purpose of identifying those circumstances that could cause a system to reach a hazardous or unsafe state. FTA is powerful static analysis tool. Given a specific hazardous state, FTA uses a *backward* (also referred as *top-down* or *deductive*) search technique in order to identify conditions that would cause the system to reach that state. In other words, once a specific hazard is identified (hypothesized), FTA will search all possible combinations of the conditions (initial states) that could force the system to reach that state.

FTA is a graphical analysis tool and uses two techniques in its analysis: qualitative and quantitative. Through the qualitative technique, FTA is capable of identifying all possible combinations of conditions that would cause the system to reach a hazardous state. These combinations of conditions are referred to as *a cut set*. A *minimum cut set* represents a minimum number of conditions that need to be satisfied in order to force the system into a hazardous state. The quantitative approach uses probability information associated with each condition (initial state) in order to calculate the probability of occurrence of the specific hazardous state. One of the advantages of the FTA is the fact that all attention is paid to a specific hazardous state and identification of preconditions that need to be satisfied in order to reach such a state. Of course, this could also become a disadvantage if FTA is the only technique used to identify hazardous states. This is due to the fact that it is possible for the analyst to overlook a specific hazardous state. In order to prevent this situation, other techniques such as Failure Modes and Effects Analysis (FMEA) [2], a *forward* (also referred as *bottom-up* or *inductive*) search, need to be used in conjunction with the FTA to identify all possible hazardous states for a system.

FTA is typically applied to hardware systems, but recently attempts have been made to apply FTA to software. Section two elaborates on previous research in the area of the Software Fault Tree Analysis (SFTA), also a road map to application of Fault Tree (FT) throughout the development life cycle is presented.

2. Software Fault Tree Analysis

There has been significant research on software fault tree analysis, with the majority having been conducted by Leveson [3], Lutz [4], and Dugan [5]. In most cases, however, SFTA is used at the code level, and the size of the software (measured by lines of code) to which the SFTA has been applied, is relatively small, approximately one thousand lines of code. Leveson [6] has generated a set of templates that could be used in SFTA, where a specific language construct (syntax) has been represented in the form of fault tree. It is important to mention that when FTA is applied to software, and specifically at the code level, we are only addressing the qualitative analysis, since at this level quantitative analysis does not make sense. Therefore, at implementation (coding phase), the objective of using SFTA is to identify the set of instructions that could possibly cause the software to reach a hazardous state. Therefore, one could use SFTA in combination with formal code inspection in order to increase their confidence in the safety of the software under investigation. Finally, it has been pointed out by a number of researchers that SFTA shows some weaknesses when there are loops involved in the code, but loops are almost always present in software. Therefore, this is a weakness that needs to be overcome. Additional work by some researchers like Helmer [7], and Modugna [8] resulted in the application of the SFTA to requirements with some success in the detection of the weak or missing requirements.

2.1 Application of SFTA during software development life cycle

Researchers and practitioners generally agree that applying SFTA at the code level is a very cumbersome and labor-intensive activity. In addition, it is a well known fact that defect detection and correction at the implementation phase is much more costly than at the earlier stages of the software development life cycle. Given this rationale, the SATC team recommends applying SFTA to requirements and design. The process is to use SFTA during the requirements and design phase to identify the critical component of the software where safety and hazardous states are the major concerns. Then SFTA may be applied at the code level only for these

critical components. The above approach follows the principle of divide and conquer, which is one of the fundamental methods of solving problems. By partitioning the system (to the safety critical component and those that are not safety critical), we narrow the scope of the area in which FTA has to be applied. Of course it is assumed that special attention is given to the flagged components (i.e., safety critical partition) during the development and verification and validation activities.

SFTA at requirements phase

The main objectives of applying SFTA during this phase of software development are to:

- Identify weaknesses that exist in the requirement specification. Weak requirements will either be modified or additional requirements will be added in order to eliminate or mitigate these weaknesses.
- Identify all the requirements that have a direct effect on the safety of the system. This can be done either through the knowledge collected as part of the requirements elicitation, or identifying the pattern of use and the surrounding environment that could affect the software, by forcing it to a hazardous state. Once requirements with safety considerations are identified, these requirements will be traced throughout the development life cycle. It is assumed that a requirement traceability matrix is included in the software development artifacts to help with this task.

SFTA at design phase

The main objectives of applying SFTA during this phase are to:

- Identify the weaknesses of the high-level design. At this stage, appropriate modifications will be implemented in order to strengthen the overall design.
- Identify the components/modules and subcomponents that have direct effect on software safety. These modules and those implementing the requirements with the safety consequences are identified. Then, special attention may be given to the generation of their implementation, by guaranteeing the elimination of design factors that could force the system into a hazardous state. The details of the application of SFTA during the design phase are discussed in Section 3 of this paper.

SFTA at implementation phase

The main objective of applying FTA to code is to identify critical code components that have direct bearing on the safety of the software. In this phase, fault trees will be generated for all the modules previously identified (during the detailed design phase) as critical modules affecting software safety. The goals here are to:

- Identifying a set of key instructions that could place the system in a hazardous state
- Add appropriate safeguards that prevent the software from reaching such a state.

As previously mentioned, the majority of the previous research in SFTA has been applicable to this phase of the software development. One of the major advantages of the above approach is to avoid generating fault trees unnecessarily for significant amounts of code in the system. It limits the application of FT to small, but critical portions of the code that affect the safety of the software. Applying FTA to the entire system requirements specification and the detailed design phase will be much more efficient than broadly applying it at the code level. Another advantage of this approach is that by applying SFTA at every stage of development, safety issues are identified early in the development life cycle and remedies can be implemented as early as possible.

3. Application of SFTA to Unified Modeling Language Artifacts

Applying SFTA during the detailed design phase will produce the best return on investment. It is here that a software product exists in its most ideal form for SFTA to be applied. Software is represented in the form of some number of modules where functionality, interfaces, inputs, and outputs are well defined. This is the closest we get to representing software structure in a way that is analogous to hardware modeling, a point prior to development where the salient system features, i.e., gates, encoder, functionality, interfaces, inputs and outputs are well defined. The same can be said about a software system at the detailed design phase. Here the software is represented with an equivalent amount of detail that we can achieve the equivalent degree of insight. Applying SFTA at this point enables us to identify modules (objects, methods, or functions) that could directly affect the safety of the system. In both the preliminary and detailed design phases, once a module or a set of modules is identified as having possible impacts on the safety of the system, additional safeguards need to be embedded into the design in order to guarantee their safe operation. It is worth mentioning again that generating fault trees for the system at this point will be a much more efficient choice than generating them during the implementation phase. With the exception of Pai's work [9] on dynamic fault trees for systems, we were unable to find any previous dynamic work that applied SFTA during the design phase.

The SATC team chose the Object Oriented Design (OOD) methodology as the vehicle for the application of SFTA at the design level. There are two primary reasons for choosing OOD:

1) Much recent software design uses OOD and the designs are implemented using OO languages, and
2) Recently many OODs use the UMLTM (Unified Modeling Language), which is standardized and commonly used by the software development community.

UMLTM uses a number of views and diagrams to describe software systems. The problem is how to relate these to the notation used in FTA. As the first step, we looked at all the different UMLTM diagrams and identified those we believe best match the SFTA. During this process, we identified the activity, sequence and state diagrams, as the first candidates for the application of SFTA. Communicating and validating critical system details becomes challenging, to say the least. This is because most end

users are not familiar with OO design artifacts such as graphs and diagrams; however, the majority of customers in the aerospace industry are familiar with hardware, they are generally comfortable with logic diagrams, which is the fundamental concept behind fault trees. Even in those rare instances where customers are unfamiliar with the concepts behind logic diagrams, it is relatively easy to achieve a comfort level with a handful of logic gates in a sequence diagram. These findings suggest that SFTA should be used not only as a verification technique for the software design, but also as a communication vehicle with customers.Our work also indicates that customers, after reviewing a fault tree, easily detect the occurrences of missing design components. By pointing out these missing components, they are actually completing the fault tree, thereby improving quality of the design as well as the ultimate system.

Initially we applied SFTA to the activity diagram. While we learned that it is possible to apply SFTA to the activity diagram, we also learned that special care is needed in order to handle any loop in an activity diagram. There is some ongoing research in this area, which appears promising; however, much work is still needed in this area. We then attempted to apply SFTA to the sequence diagram, at which point we came across additional findings. We learned that while SFTA may serve as a technique for verification of design, it could also serve as a vehicle for improved communication with customers and other stakeholders. We have developed a partial paradigm for transforming sequence diagrams into software fault trees. Ultimately, we applied SFTA to the state diagram. We arrived at the same set of observations as in the case of sequence diagrams. Figure 1 represents the state diagram for a pay at the pump system, with its corresponding fault tree diagram represented in Figure 2. As noted for activity and sequence diagrams, special care must be given when representing timing constraints and occurrences of iteration.

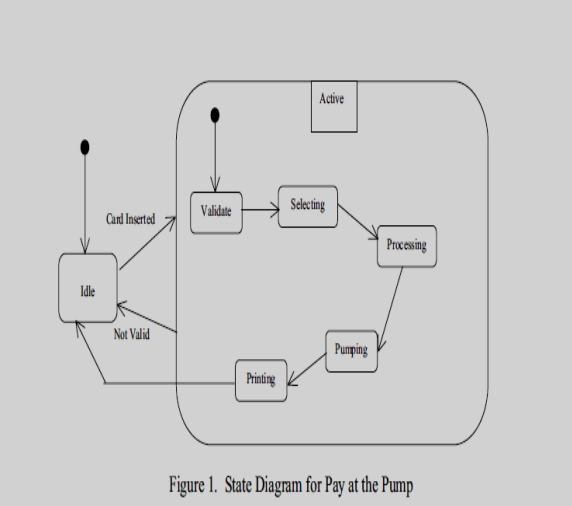


Figure 1. State Diagram for Pay at the Pump

4. SFTA FOR USE CASE BASED REQUIREMENT ANALYSIS

There exists very little published work regarding the application of SFTA in use case based requirement analysis process. In use-case based fault tree analysis work, the major functionalities carried by use cases are derived first and afterwards fault tree is drawn manually for the failure scenarios for such functionalities. Douglass's work reported in has stressed upon first drawing the fault tree and then link the child nodes of the fault tree with the suitable use case functionality. The work regarding the functional hazard assessment (FHA) of use cases via integrated application of a technique named functional failure analysis (FFA) and fault trees.

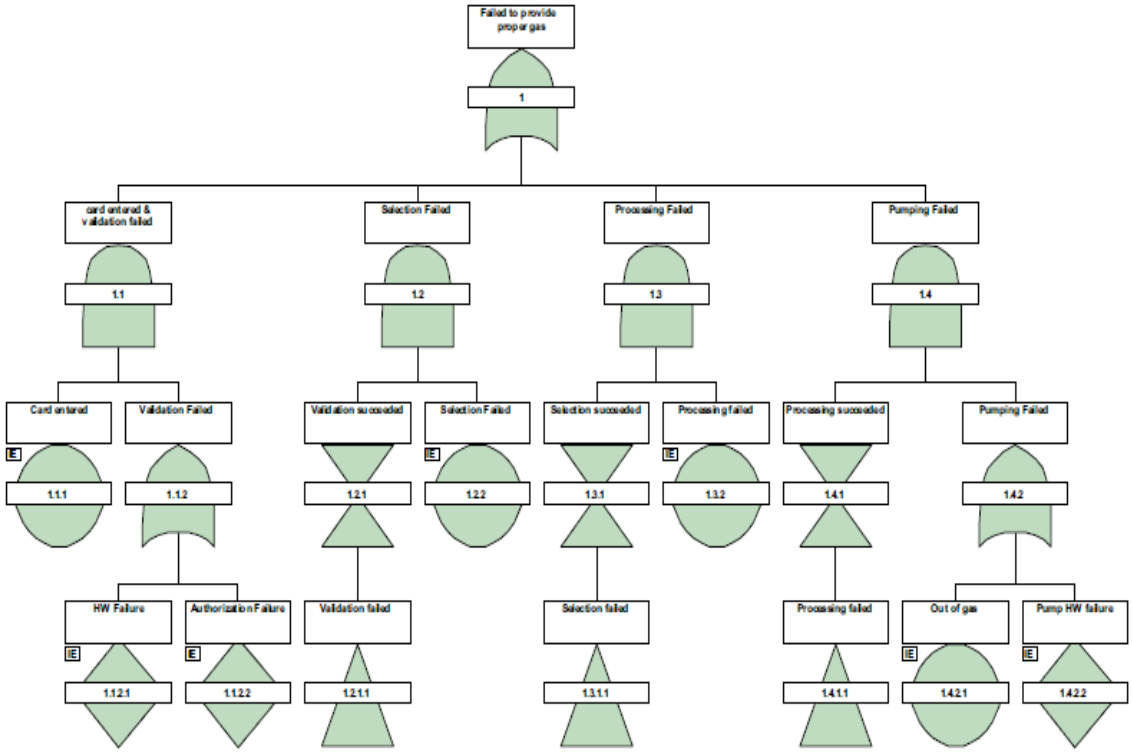


Figure 2. Fault Tree for Pay at the Pump

5. PROPOSED SFTA APPROACH FOR UCRT

The work regarding how to apply SFTA on the textual description of use case in the form of UCRT is missing from literature and the approach presented in this section will try to fill that gap. The proposed approach will take UCRT of various use case scenarios as input. The major requirement of the presented approach is that hazard should be expressed in the form of incompatible states of its components. For example, consider a software system which controls the functioning of two components say X (with possible states *x1, x2, x3*) and Y (with possible states *y1 & y2*). Now hazard state can be expressed as *X=x1 and Y = y2* if and only if this combination of states of X and Y is found to be serious and dangerous. The state change matrix of UCRT will be used to identify those action steps which can contribute in the occurrence of the selected root hazard. Each and every event and action step mentioned under normal flow of events of every UCRT has been assigned a logical time stamp in increasing order. This logical time stamp will help in identifying the component whose transition occurred first than the other.

Proposed Approach

Input: UCRT of use cases & hazardList = {list of hazard states}

Assumptions:

1. Hazards are expressed using states. [For example elevator = moving and door=open].
2. Hazard can occur only during the execution/realization of use case action steps.
3. The fault tree will be drawn only for those action steps from UCRT which can contribute for the root hazard.
4. Timing related hazards have not been considered for this approach and that's why each action step has been assigned a logical time stamp rather than physical time.

FOR each hazard state in the hazard List
DO
(i) Identify the UCRTs which can contribute in above hazard.
(ii) IF number of UCRTs associated with the selected hazard state is more than 1 THEN
(a) For each UCRT identified in step (i) above draw the fault tree after identifying the action steps from that UCRT which can contribute in hazard state.
(b) Combine the fault tree(s) created above steps by an OR gate and feed the output to root event
(c) Add any hardware related erroneous event if it can independently cause the hazard
ELSE
Draw the fault tree from the steps of single use case using state change matrix
(d) Determine the minimal cut sets for the fault tree created above and derive safety requirements from minimal cut sets.
ENDIF
ENDDO
ENDFOR

Description of the Proposed Approach

The operational details of the above mentioned approach has been descried below by taking a hypothetical hazard
X='a' and Y = 'b' as an example. (*Note that X and Y are components where as 'a' and 'b' are states*).

(a) How to identify UCRTs which can contribute in the selected hazard X = 'a' and Y = 'b'.

Scan the state change matrix of each UCRT for the entry X='a' and if found then within same UCRT search for Y ='b'. If found mark that UCRT otherwise ignore that UCRT (Note that *X = 'a' and Y = 'b'* may appear in different rows).

(b) How to identify the action steps from the selected UCRT which can contribute in the hazard *X='a' and Y='b'*.

(i) Scan the UCRT to find out the first occurrence for the entry *X ='a'* and note down the logical time '***t1***' for that action step. Scan the UCRT to find out the first occurrence for the entry *Y ='b'* and note down the logical time ***'t2'*** for that action step.

(ii) If ***t1*** is less than ***t2*** then scan UCRT to note down the logical time ***'t3'*** of the last action step when ***X='a'*** and note down the events and actions that occurred between time ***t3*** and ***t2***. Otherwise (when t2 < t1) scan UCRT to note down the logical time ***'t3'*** of the last action step when ***Y='b'*** and note down the events and actions that occurred between time ***t3*** and ***t1***.

(c) How to draw the fault tree for the selected action steps of UCRT.

If transition of the state of component X from state 'a' to any other state occurs earlier in UCRT than transition of state of component Y from state 'b' to any other state then it means hazard can occur if state transition of X from 'a' has not occurred and
Y has successful made transition to state 'b'.
Otherwise the other alternative for the occurrence of the root hazard may be that the transition of Y from state 'b' has not occurred at all and X has successful made transition to state 'a'. Identify the erroneous events which prevent the state transition from occurring for that component whose logical time (t1 & t2 noted above) is less and feed them to an appropriate OR gate. Identify the events and actions which changes the state of that component which makes transition at a later point and feed these events to an AND gate. In the end the out put of previous OR gate should be feed to the AND gate to complete the fault tree.

5. SFTA APPLICATION FOR ELEVATOR CONTROL APPLICATION

The TABLE II and III indicate the UCRTs for a *select destination use case when elevator is idle* and *when it is moving*. The state change matrix of both the tables records the states of three components elevator, motor and door. An elevator can be in any of the five possible states as *{idle _ prepare_to_move _ moving _ prepare_to_stop_at_floor _ at_floor}*, motor can be in any of the two possible states *{started _ stopped}* and door has only two possible states *{open _ closed}*. When the elevator is idle it remains on the last visited floor with door open. When elevator is in moving state then door should not be in open state and similarly elevator = idle and door = closed combination state should exists only for a specified period of time. The elevator door and motor are synchronous devices in the sense that they report back their status (open or closed in case of door, started or stopped in case of motor) and their working is strictly controlled by elevator controller

application. The elevator button and floor buttons are asynchronous devices and they only notify the system when ever user presses either the floor or elevator button and system will respond suitably by carrying out the desired action.

SFTA Application for UCRTs of TABLE II and III

Step1. hazardList = {***elevator = 'moving' and door ='open'***
(elevator is in moving state and door is in open state), ***elevator = 'at_floor' & door ='closed'*** (door does not open on reaching the floor)}.

Step2.

(a) Drawing Fault Tree for the hazard *elevator = moving and door = open*.

(i) Identification of use cases which can contribute in the above hazard.

The careful observation of state change matrix of each identifies UCRTs of TABLE II and TABLE III as both contains entries for *elevator = moving and door = open*.

(ii) Drawing of Fault Tree for the selected UCRTs.

Drawing Fault Tree from TABLE II.

At the action step 2(iv) with logical time 8 of TABLE II, elevator changes its state from idle to moving. Similarly at action 2(ii) of TABLE II with logical time 6, door changes its state from open to close. It clearly indicates that door has changed its state earlier than elevator and therefore the possibility is that door state has not made the transition from open to closed where as elevator state has successfully make the transition from idle to moving. The action steps that can force the system in to hazard state are {2(ii), 2(iii) & 2(iv)}. The errors that can occur at step 2(ii) are: *system fails to give the door close command* (Event E1 of Figure 3) or *door does not close upon system command* (Event E2 of Figure 3). Similarly error that can occur at step 2(iii) is: *system wrongly determines the status of door as closed where as it is still open* (Event E2 of Figure 3). The error that can occur at step 2(iv) is that *system issues motor start command when door is still open* (Event E2 of Figure 3). The fault tree drawn from TABLE I for hazard *elevator = moving and door = open* is in Figure 3.

TABLE I. Basic Fault Tree Events and Logic Gate Symbols

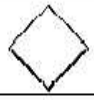
Fault Tree Event Symbols		
Events	Symbol	Description
Basic Event		A basic initiating fault (or failure event). It is a basic event that does not need further resolution.
Undeveloped Event		An event which is not further developed.
Conditioning Event		A specific condition or restriction that can apply to any gate
Top Event		An event to be analyzed. i.e to event. It can also be used for intermediate event.
Fault Tree Logic Gate Symbols		
Logic Gates	Symbol	Description
AND		The output event occurs if all input events occur
OR		The output event occurs if at least one of the input events occurs.
Priority AND		The output event occurs if all input events occur in a specific sequence / specified priority

TABLE II: UCRT for Select Destination Use Case when Elevator is idle

Use Case Name			Select Destination (When Elevator = idle)		
Actor			Elevator User		
Precondition			User is in elevator and elevator = idle and door = open, motor = stopped		
PostCondition			Elevator should visit the floor selected by the user		
Normal Flow of Events			**State Change Matrix**		
S.No	**Actions**	**Logical Time**	**Elevator = idle**	**Motor= stopped**	**Door= open**
1	WHEN user press elevator button THEN	1	*idle*	*stopped*	*open*
(i)	elevator button sensor reads the destination floor request and notifies it to system	2	*idle*	*stopped*	*open*
(ii)	system update the request	3	*idle*	*stopped*	*open*
2	IF elevator = idle THEN	4	*idle*	*stopped*	*open*
(i)	Determine direction	5	*idle*	*stopped*	*open*
(ii)	system commands the door to close	6	*prepare to move*	*stopped*	*closed*
(iii)	WHEN door = closed THEN	7	*prepare to move*	*stopped*	*closed*
(iv)	start the motor	8	*moving*	*started*	*closed*
3	As the elevator is approaching the floors, floor sensor detects that and notifies the system	9	*moving*	*started*	*closed*
(i)	System checks if elevator has to stop at that floor	10	*moving*	*started*	*closed*
(ii)	IF yes THEN	11	*moving*	*started*	*closed*
(iii)	System commands the motor to stop	12	*Preparing to stop at floor*	*stopped*	*closed*
(iv)	WHEN elevator = stopped THEN	13	*Preparing to stop at floor*	*stopped*	*closed*
(v)	System commands the elevator door to open	14	*at_floor*	*stopped*	*open*

TABLE III: UCRT for Select Destination Use Case when Elevator is moving

Use Case Name			Select Destination (When Elevator = moving)		
Actor			Elevator User		
Precondition			User is in elevator and elevator = moving and door = closed, motor = started		
PostCondition			Elevator should visit the floor selected by the user		
Normal Flow of Events			**State Change Matrix**		
S.No	**Actions**	**Logical Time**	**Elevator = moving**	**Motor= started**	**Door= closed**
1	WHEN user press elevator button THEN	1	*moving*	*started*	*closed*
(i)	elevator button sensor reads the destination floor request and notifies it to system	2	*moving*	*started*	*closed*
(ii)	system update the request	3	*moving*	*started*	*closed*
2	As the elevator is approaching the floors, floor sensor detects that and notifies the system	4	*moving*	*started*	*closed*
(i)	System checks if elevator has to stop at that floor	5	*moving*	*started*	*closed*
(ii)	IF yes THEN	6	*moving*	*started*	*closed*
(iii)	System commands the motor to stop	7	*Preparing to stop*	*stopped*	*closed*
(iv)	WHEN elevator = stopped THEN	8	*Preparing to stop*	*stopped*	*closed*
(v)	System commands the elevator door to open	9	*at_floor*	*stopped*	*open*

Drawing Fault Tree from TABLE III

Similarly the fault tree from TABLE III for hazard *elevator = moving and door = open* can be drawn and is shown in Figure 3. It should be noted that in TABLE III elevator state is changing first from moving to preparing_to_stop_at_floor where as door is changing its state from closed to open at a later point in time. There is a possibility that elevator state may not change whereas door state may change and hence can lead to a hazard.

Combining fault Trees for both use cases

The fault trees drawn from TABLE II & III are combined via top level OR gate as shown in Figure 3.

Adding any other event if required.

The error event B3 has been added to complete the fault tree. It is basically an independent hardware failure and can happen at any point of time and has the potential to cause the root hazard alone. The complete fault tree the hazard ***elevator = moving and door = open*** is shown in the Figure 3.

(b) Drawing Fault Tree for the hazard elevator = at_floor and door = closed.

The fault tree for the hazard *elevator = at_floor and door = closed* is shown in Figure 3.

(c) Determine the minimal cut sets and derive safety requirements

The minimal cut sets for the fault tree of the Figure 3 are as follows:

(E3 * D4 * C4) + (E4 * D4 * C4) + (E1 * D2 * C2) + (E2 * D2 * C2) + (B3)

(Note * means AND whereas + means OR).

For ***AND*** sequence of events the safety requirements can be provided for any one of the events whereas for ***OR*** sequence of events the safeguard will have to be provided for every event. The events E1, E3, D2, D4, C2 & C4 are erroneous events related to software failure where as E2, E4 & B3 are erroneous events related with hardware failures where as rest are intermediate events. Now it is known that how top event can occur and what logical combination of events can cause it, so safety requirements can be derived for the most important ones. For example consider the first sequence of events {E3 * D4 * C4}. The event D4 is most important one and if safeguard against it is provided in the system, then the risk related with the occurrence of root node can be minimized. The event D4 is: ***system check determines the motor as stopped where as it is still moving***. There are two ways to deal this event- either allows it to happen and then detect the error and provide a additional level safeguard for it or make necessary safe guards so that it should not occur. The first approach is generally followed i.e. use a monitoring object to observe the working of elevator controller application which can take necessary actions i.e. activation of emergency shutdown mechanism in case such event occurs. This way of providing a safe guard is an example of Monitor-Actuator design pattern. The same safe guard can be applied for event D2. The safeguards for D4 & D2 eliminates the first four sequences {(E3 * D4 * C4) + (E4 * D4 * C4) + (E1 * D2 * C2) + (E2 * D2 * C2)} from the minimal cut sets. Now only one event B3 which is an independent hardware failure is left which can still cause the root hazard. The safeguard against it will be to provide additional safety features in the elevator controller application itself at design time – which will detect the malfunction of door and activates the emergency shout down mechanism or stopping the elevator to a nearest floor.

Similarly minimal cut sets for the tree of Figure 3 are X + Y. Since X is software control error event – the use of controller object via an actuator-monitor design pattern will be suitable and Y is an independent hardware failure – so this malfunctioning should be detected by the controller application and suitable response action should be activated.

6. CONCLUSION

The effective strengths of SFTA in indentifying missing and additional safety requirements from use case textual description expressed in natural language has been demonstrated in this paper. The main weakness of SFTA which has been observed in the past by various researchers also is its tedious and time consuming nature. SFTA being a backward safety analysis technique can not sometimes identify all erroneous events and that's why to achieve better results SFTA application should be integrated with forward safety analysis technique such as SFMEA as has been explored by Lutz

REFERENCES

1. NUREG-0492, Fault Tree Handbook, U.S. Nuclear Regulatory Commission, January, 1981.
2. "Applying Integrated Safety Analysis Technique (Software FMEA and FTA)", Jet Propulsion Laboratory, November 1998.
3. Leveson, N. G. and P. R. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983, pp. 569- 579.
4. Lutz, R. R., "Targeting Safety-Related Errors During Software Requirements Analysis," *Journal of Systems and Software*, 1996, 34, 223-230.
5. Dugan J.B., Sullivan, K. J., and D. Coppit, "Developing a High-Quality Software Tool for Fault Tree Analysis", *Transactions on Reliability*, December 1999, pp. 49-59.
6. Leveson, Nancy G., Stephen S. Cha, Timothy J. Shimeall, "Safety Verification of Ada Programs Using Software Fault Trees." *IEEE Software*. pp 48-59.
7. Helmer, G., Slagell, M., Honavar, V., Miller, L. and Lutz, R., ″A Software Fault Tree Approach to Requirement Analysis of an Intrusion Detection System″ *Symposium on Requirements Engineering for Information Security,* March 5-6, 2001
8. Francesmary Modugno, Nancy G. Leveson, Jon D. Reese, Kurt Partridge and Sean D. Sandys, `Integrated Safety Analysis of Requirements Specifications", *IEEE International Symposium on Requirements Engineering*, 1997.
9. Pai, G., J., and J. B. Dugan, "Automatic Synthesis of Dynamic Fault Trees from UML System Models", *The 13th International Symposium on Software Reliability Engineering*, IEEE Computer Society, Annapolis, MD, USA, November, 2002, pp. 243- 254.
10. OMG Unified Modeling Language, Version 1.4, September, 2001, http://www.omg.org/cgibin/doc?formal/01-09-67